RL-TR-93-164
Final Technical Report
August 1993

NPAC DISTRIBUTED-PARALLEL SYSTEM FEASIBILITY STUDY

Syracuse University

Dr. Nancy McCracken, Dr. Gary Craig, Don Hewitt, and Kanchana Parasuram

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-93-164 has been reviewed and is approved for publication.

APPROVED:

JON B. VALENTE
Project Engineer

FOR THE COMMANDER

JOHN A. GRANIERO
Chief Scientist for C3

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3AB) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1993	Final Sep 90 - Sep 91

4. TITLE AND SUBTITLE

NPAC DISTRIBUTED-PARALLEL SYSTEM FEASIBILITY STUDY

5. FUNDING NUMBERS

C - F30602-88-D-0025,
PE - 62702F Task 0045
PR - 5581
TA - 21
WU - P1

6. AUTHOR(S)

Dr. Nancy McCracken, Dr. Gary Craig, Don Hewitt,
and Kanchana Parasuram

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Northeast Parallel Architectures Center
Center for Science and Technology
Syracuse University
Syracuse NY 13244-4100

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory (C3AB)
525 Brooks Rd
Griffiss AFB NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-93-164

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Jon B. Valente/C3AB/(315) 330-3241
Prime Contractor is Georgia Institute of Technology.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

This study established a distributed-parallel computing testbed, comprised of two shared-memory multiprocessors linked by a wide-area network. A missile-tracking simulator application was implemented on this testbed for the purposes of evaluating the feasibility of distributed-parallel computing. Several scenarios were run and performance data collected for each of four computing modes. Preliminary evaluation of the software model and analysis of the scenario performance data are presented.

14. SUBJECT TERMS

Distributed System, Parallel Processing

15. NUMBER OF PAGES

28

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NPAC Distributed-Parallel System Feasibility Study

Georgia Institute of Technology
Contract F30602-88-D-0025

Final Technical Report

Dr. Nancy McCracken
Dr. Gary Craig
Don Hewitt
Kanchana Parasuram

Northeast Parallel Architectures Center
at Syracuse University

Introduction

Advances in the use of both distributed computing systems and in parallel computers have led to the consideration of using a distributed-parallel computing system, which is a distributed system of computing nodes in which some of the nodes may be parallel computers. Such systems can combine attributes of both systems such as high performance, redundancy and reliability, and physical proximity of parts of the computing to the data.

One use of a distributed-parallel system is in the enhancement of an ordinary distributed system to one in which one or more of the nodes may be high performance computers. Current projections of high performance computing architectures are that these machines will necessarily be parallel computers. Examples of applications which will require this sort of distributed-parallel system are those in which large numbers of small computing nodes which may service requests or collect data may then send those requests or data collections off to a large computing node for processing. Although these applications add the issue of heterogeneity to the distributed computing model, this issue has been addressed by a number of systems without otherwise change to the underlying distributed processing model.

The other use of a distributed-parallel system is in the enhancement of a single parallel machine to the parallelism achievable by many (possibly parallel) nodes in a distributed system. Essentially, this idea is to scale up the parallelism found in a single machine, where the communication times are relatively fast over a bus or network, to a parallel system where the communication times are relatively slow over a local area network (LAN) or wide area network (WAN). In addition to enhanced computing power, this use of a distributed system could add the distributed system capabilties of reliabilty and redundancy, etc. to a high performance system. This use of a distributed processing system may change the underlying processing model since using the distributed processes as parallel processes implies a tighter coupling of the data and process interactions as is found in parallel processing models. More investigations are needed in this area to understand the parameters that make this kind of distributed-parallel computing feasible.

Scope of work

The goal of this project is to demonstrate the feasibility of distributed-parallel computing in the second case (as defined above) by implementing a parallel application on a distributed system of parallel computers and designing evaluation criteria. The project is designed as a sequence of four tasks as described in the GIT contract:

1. Establish a working distributed-parallel platform between the Northeast Parallel Architectures Center (NPAC), located in Syracuse, New York, and Rome Laboratory,

located in Rome, New York, using the Cronus Distributed Operating System.*

2. Design and specify a distributed-parallel application which can exploit the capabilities of a diverse set of parallel architectures where the entire computation must be coordinated over a set of individual computational nodes.

3. Perform analysis to determine an appropriate criteria to evaluate the performance of a distributed-parallel system.

4. Based upon recommendations from Task 3, a candidate demonstration will be specified by which the capabilities of the system developed in Task 1 can be shown.

The remainder of this report discusses the implementation and the results of each of these tasks.

The Distributed-Parallel Computing Platform at Rome and Syracuse, NY

The establishment of the distributed-parallel computing platform made use of available hardware and software at Rome Laboratory in Rome, NY, and NPAC at Syracuse University in Syracuse, NY. Although the design and evaluation phases of this project were careful to plan for a more general case in which the parallel application may be implemented on a heterogeneous distributed system, this implementation was chosen to be carried out on a homogeneous system consisting of two Encore Multimaxes**. Both of these machines were shared memory MIMD Encore Multimax 320's. The one at Rome has 16 parallel processors and the one at Syracuse 20 parallel processors.

Both of these Multimaxes are on local area networks at their respective sites with Sun workstations (and other hardware not used in this project) and a gateway to the WAN run by Nysernet. Between the gateways of Rome Laboratory and Syracuse University, a distance of about 50 miles, is a T1 line with no other intermediate gateways (see Figure 1).

In addition to these machines, the hardware for this project included two Hewlitt-Packard 4972A Local Area Network Protocol Analyzers (LANPAs). These were provided by Rome Laboratory. One was installed in the Rome Laboratory LAN and the other in the Syracuse University LAN. These devices are able to monitor the actual message packets of any communications between the two Multimaxes as the packets go out and come in through the gateways.

The distributed operating system chosen for this platform was the Cronus distributed programming environment from BBN. While this system is widely available as a heterogeneous distributed system, this project was one of the first to make use a new

---

* Cronus is a product of BBN Systems and Technologies Corporation

** Multimax is a trademark of Encore Computer Corporation

extension to enable distributed processes to also take advantage of running on a parallel machine like the Multimax. In fact, this was one of the uncertainties of the project. Originally, it was planned to run Cronus on the Umax operating system and this platform was established at Rome on the Multimax 320 and at Syracuse, the Multimax 520 (which differs essentially in having faster processors). However, the parallel computing support was only available in Cronus under the Mach operating system, and the platform was changed accordingly. Rome Laboratory acquired Mach and installed it with Cronus on their Multimax 320, and NPAC installed Cronus on the Mach already running on their Multimax 320.*

Another part of the software platform was the code written to collect message timings from the LANPAs. The code to analyze packets on the LANPAs was written by the team from Rome Laboratory and the code to sychronize clocks between the LANPAs and the Multimaxes was written by a systems administrator from NPAC and Rome Laboratory.

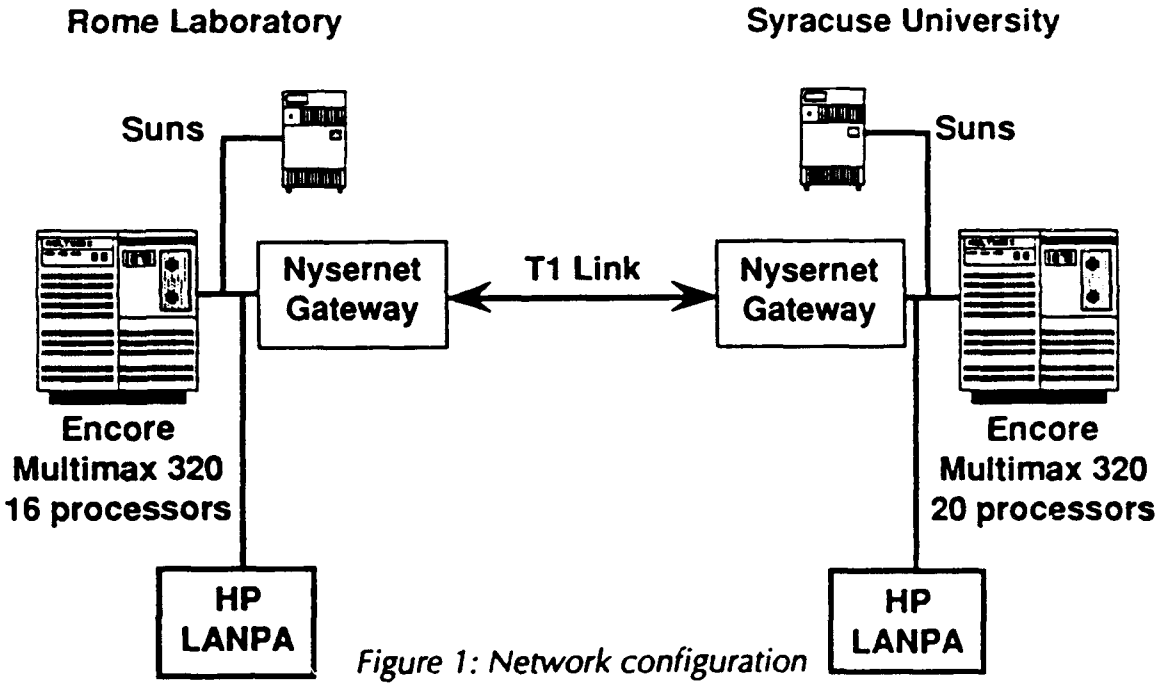


Figure 1: Network configuration

The Experimental Application

The first step in choosing an experimental application was to consider the characteristics of an application that could take advantage of this kind of distributed-parallel platform. During the consideration of characteristics of applications, both the team from Rome Laboratory and the team from NPAC studied the distributed and parallel process models. Assistance was given by training from Rome Laboratory about the Cronus distributed system and from NPAC about parallel processes in Mach.

* We are grateful to the support given both by Encore Corporation for providing Mach and to BBN for providing the parallel process implementation in Cronus.

Desired Characteristics

The characteristics which were found to be appropriate were those concerning the computation to communication ratio and those of system-wide synchronization.

A suitable application must have sufficient granularity of parallelism to take advantage of the combined total of 36 processors. It must not have so large an amount of communication that the advantages of using two parallel computers is lost. Furthermore, it must not have communication in a synchonization pattern such that the advantage of using two parallel computers is again lost. Finally, we decided against using an application with essentially no communication, what is called an "embarassingly parallel" application where the parallel subprocesses are totally independent of each other. This kind of application would indeed take advantage of two distributed parallel processors, but we preferred to concentrate on the more challenging case of an application where the parallel subprocesses must have some degree of communication between them. This would lead to an experiment that would show whether an even larger class of parallel applications than the "embarrassingly parallel" ones would be feasible on this system.

Multisource Tracking Simulation - Overview

The application chosen was a multisource missile tracking program originally written at Caltech in conjunction with JPL for the Hypercube machine. [Gottschalk 1987] This program is publicly available as part of the Caltech benchmarking suite. [Messina et al 1990] The tracking program receives a set of missile coordinates at each time step and establishes a set of tracks based on calculated trajectories. The tracking program is intended to be part of an application in a distributed system of sensors observing the missile data, which is communicated to a large processing node running the tracking program. Our experiment implements the large processing node as two parallel machines, and the sensor data is generated and controlled by a subroutine.

The structure of the program is a large loop based on time; the loop body is executed once for each set of sensor data (see Figure 2). The main data structure of the program is the current set of tracks being determined. The loop body consists of using the sensor data to sequentially
1) extend the "current" tracks using a rough filter (which adds all possible extensions to the tracks file)
2) a precision filter that deletes bad extensions, and
3) initiating any new tracks.
Most of the computational time is spent in the precision filter; all of these computations can be parallelized over the tracks.

We had several versions of the program to work with: 1) a parallel version developed for a distributed memory MIMD machine, 2) a sequential version derived from the first,

and 3) a parallel version adapted from the second for a shared memory MIMD machine. The original code had been parallelized by dividing up the current set of tracks among all the processors. The precision filter code is not completely indepedent in each track, but depends only on other nearby tracks (primarily to detect false duplicate tracks). During each loop body, tracks are created and deleted. Tracks must be moved between the processors to: 1) colocate possible duplicate tracks and 2) insure that large differences in the number of tracks on each processor do not occur. Thus a load balancing phase, in which tracks will be more evenly distributed among the processors is added into the main loop body just prior to the precision filter computation.

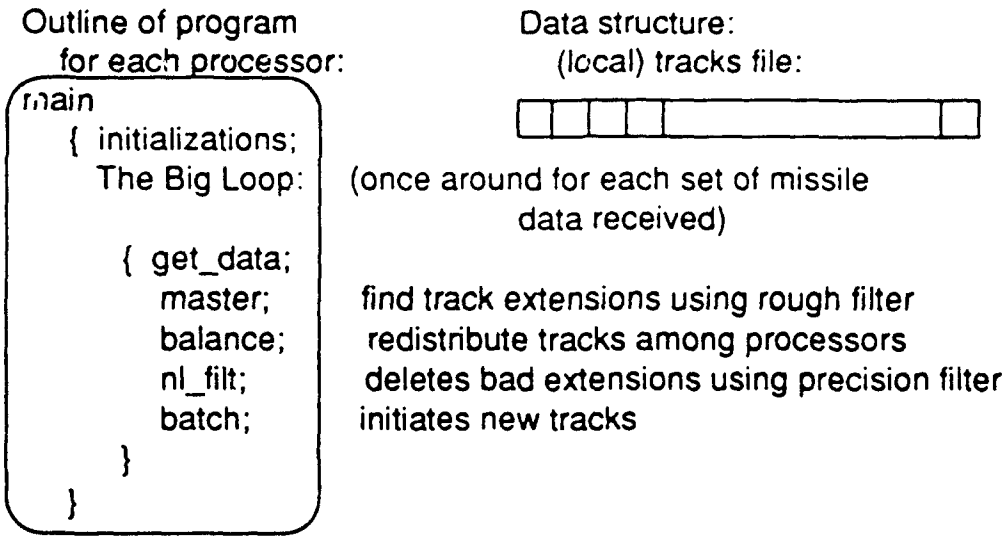


Figure 2: Application Structure

This application fits our criteria for a distributed-parallel system. There is plenty of parallelism since the number of tracks will be in the hundreds for even fairly small problems. The amount of computation is large with respect to the amount of communication. Although the distributed processors must synchronize during the communication of load balancing, this only happens once during each loop body.

Programming Model

An important aspect of this project was to examine several different computational models: sequential processing, parallel processing, distributed processing and distributed -parallel processing. In order to facilitate this, Cronus, a distributed computing environment developed by Bolt, Beranek and Newman Inc. (BBN), was chosen as our development environment. Cronus utilizes an object-oriented client/server programming model. Cronus' strengths lie in its support for a number of different platforms (heterogeneity) and support for fault-tolerance.

In Cronus, objects are passive and are "managed" by a multi-threaded manager (server). As such, each manager represents a shared address space for a set of common typed objects. This maps well to either a uniprocessor or a shared-memory multiprocessor. New threads of control are automatically generated to handle an operation invocation (request) on the server. In addition, new threads may be explicitly created.

If several threads of control were ultra-lightweight and a server had a very large buffer for pending operations, a very general programming model develops. This model is one in which the managed objects are tracks and a simulation control task (associated with each manager) generates asynchronous operation requests for each track in the local tracks "database". The result, on different architectures, would be sequential processing on a uniprocessor, and N-way multiprocessing on a N-node shared memory multiprocessor.

As will be noted in detail later, experimentation determined that no thread is infinitely "lightweight" and there are real limits to the buffering capacity of a server. Thus, the application was coded to uniformly distribute tracks to N explicitly created threads on an N-node processor.

Partitioning

One manager, Tracks Manager, was implemented in C and instantiated once on each distributed node (although we implemented this code on two nodes, the code was written more generally). Cronus manages machine dependancies and permits the code to be portable to a number of different platforms. Within each manager, the code is parallelized over the tracks assigned to that node (see Fig. 3). The major programming tasks were to rewrite the load balancing to use Cronus message-passing (by invoking a method in another manager), and in implementing the shared-memory parallelism within each distributed node.

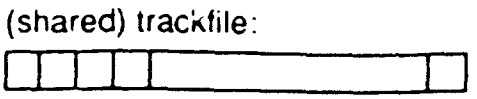
In implementing the parallelism within each distributed node, we used an extension of Cronus in which the distributed process model was extended to a parallel process model. In distributed process model, multiple invocations of operations (methods) in a manager were shared memory concurrent processes. But, each of these processes was implemented in a coroutine fashion on a single processor. Under this implementation, there were no synchronization problems with using shared memory since each process was guaranteed to run to completion or until it yielded the processor.

The extention to allow truly parallel execution of the processes within one manager has processes request to run on their processsor. Hence, multiple invocations of operations can run many processors. However, now if more than one procedure may modify some data in the shared memory, synchronization must be used to ensure data integrity. Semaphores are available for this synchronization.

```
Manager1:

master:
   for all tracks
       Extendtrack(trackid);

where
   Extendtrack(x)
   {   Getownprocessor;
       ...
       Releaseownprocessor;
   }
```



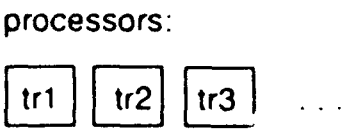


Figure 3: Form of parallelism in each manager

The general decomposition of "parallel" operations is demonstrated by example in Fig.3. The original code would sequentially execute an operation, e.g., extendtrack, for each track (in the managers trackfile). The parallel code creates an independent task (thread) for each operation where the operation code explicitly asks for independent sheduling via the TaskObtainOwnProcessor () call.

The Tracks Manager exports two distinct interfaces (sets of operations). The first interface is used by the sensor data generator. There are three operations available in this interface: NewSimulation, SimInit, and AcceptData. NewSimulation is used to determine the availability of a Tracks Manager (only one such manager can exist on a host and it was coded so that only one simulation could be active at a time). SimInit is used to pass general simulation data (initialization) to a manager, e.g. number and location of sensors. Finally, AcceptData is invoked for each new sensor scan to pass the sensor data to the Tracker.

The second is the one which enables managers from multiple hosts to cooperate on a single simulation. These are the operations used to implement the (load) balancing phase. This interface has two operations: AcceptTrackList and AddTrack. AcceptTrackList is used to transfer a summary of a manager's local list of tracks to other managers. This information is needed so that each manager can autonomously determine which tracks must be transfered and to which manager. AddTrack is the interface which transfers a block of tracks from one manager to the other (see Fig. 4).

Distributed-Parallel Software Lessons

Even though it was not a main goal of our project to evaluate a distributed-parallel software environment in general, or Cronus in particular, since we did use Cronus for this project, a discussion of the problems and successes in using this kind of software may be helpful to software designers.

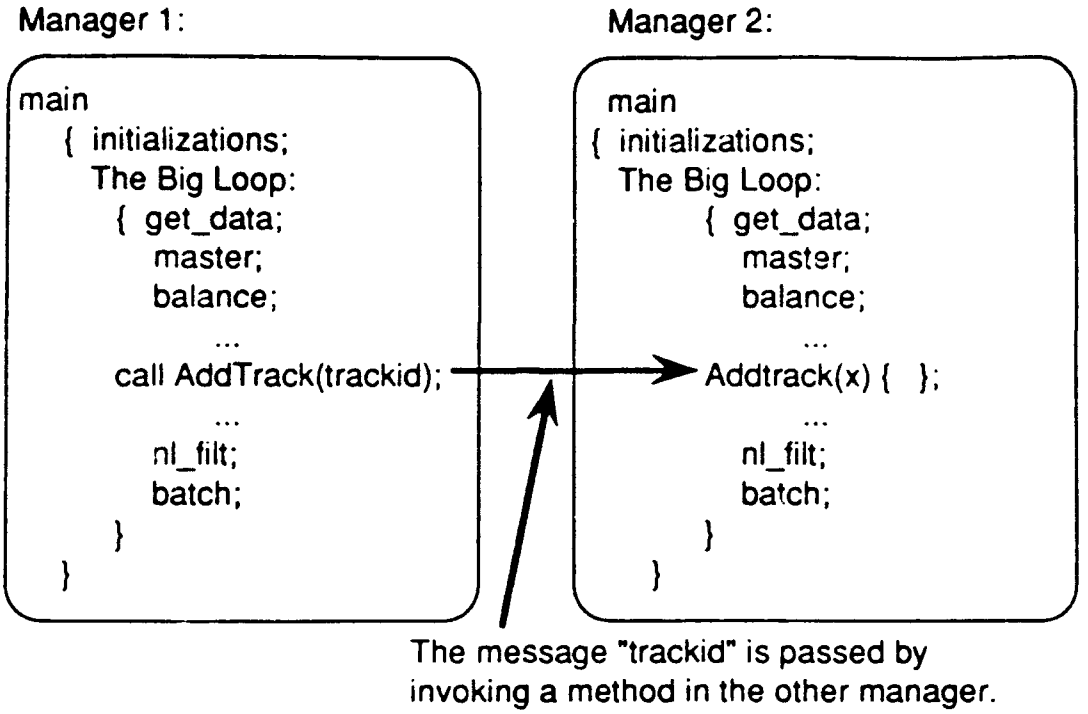


Figure 4: Message passing between managers

The advantages of using a high-level software system like Cronus, instead of a low-level system like Mach or Express, are great. We felt that the time spent porting code was very small due to the tools for managing a group of distributed processes and the simple message-passing model. It was also easy to design for a more general system than we actually had; it was easy to structure the code for arbitrary numbers of nodes and hetereogeneity is built in. We also felt that the extention of Cronus to include parallel processing nodes was successful on the shared memory parallel processors.

Of course, we must expect to pay some overhead for using a high-level heterogeneous software system, but there were a couple of areas where we felt that the overheads were unacceptably large. Where possible we modified the software design so as to better reflect the needs for our analysis. In other words, where possible we didn't want our results to simply reflect constraints imposed by our design environment.

Our first design of the code was a standard Cronus, persistant object-oriented code design, in which tracks were objects within the managers. However, Cronus keeps these in a persistant database, which is appropriate for applications in which reliability and redundancy are most important. In our application, with rapidly changing data as tracks are added and removed, the overhead of persistant objects was prohibitive and we made them into an ordinary volatile data structure.

Our initial pass at implementing parallel threads was to take advantge of the implicit task creation which occurs during an object invocation on a manager. In this scenario, a manager would have a coordinating task (executing the main loop), which would issue

asynchronous invocations upon the manager for each "track". Under this design, if a manager could allocate multiple processors for incoming invocations (in the case of a uniprocessor), it would just queue up the pending invocations and serve them sequentially. In fact the TaskOwnObtainProcessor () nicely handles this situation in a way that the manager code could be identical on both a uniprocessor and a multiprocessor.

Two separate performance issues required this solution to be redesigned. First, operation invocation (and the subsequent task creation) is a relatively expensive activity (as implemented in Cronus on the Mach OS on the Encore Multimax). Thus some operations, which were independent over tracks, could not effectively be parallelized since the invocation time dominated the actual processing time. Second, in any client-server (message-passing) environment, message queues are implemented as finite buffers. We found that we could easily overrun the message queue buffer. Messages were lost if we did not explicity program in a wait after a certain number of messages were sent. This is a common problem in message-passing systems and one which high-level software paradigms should address.

In the final implementation, a single invocation was made to a parallel code block. The code then checks to determine the number of available processors and creates up to that number of parallel tasks, and correspondingly partitions up the track file to each task (data parallelism). The resulting code was still portable between both uniprocessors and multiprocessors. In addition, the partitioning code could amortize the cost of task creation over the amount of computation each task would perform. The result was significantly better parallel performance.

Performance Evaluation Methodology

The most important aspect of performance evaluation of a distributed-parallel system is its system-wide performance, as opposed to single parallel processor performance or distributed operating system performance. This unit of measure is essentially that of elapsed time of the implementation and can be compared with the other modes of computing.

System-wide comparisons which can be made for any distributed-parallel application are:

- Sequential — running the application on one node with one processor.
- Parallel — running the application on one node with many processors.
- Distributed-parallel — running the application on more than one node where each node may run many processors.
- Distributed — running the application on more than one node where each node only runs one processor.

In addition, if feasible, one can compare the effects of nodes of local area networks versus wide area networks. In our case, our local area network machines were Sparc workstations, so that our comparisons had to factor in relative cpu speeds. We also had

Analysis Objectives	Tests							
	One node					Two nodes		
	1	2	3	4	5	6	7	8
Cronus overhead for organization	x	x	x	x				
Sequential vs. Distributed LAN	x						x	
Sequential vs. Distributed WAN				x				x
Sequential vs. Parallel				x	x			
Parallel vs. Distributed-parallel					x	x		
Distributed vs. Distributed-parallel						x		x

1. Sequential SPARC
2. Sequential Multimax
3. Sequential SPARC Cronus
4. Sequential Multimax Cronus
5. Parallel Multimax Cronus
6. Dist.-parallel Multimax Cronus
7. Distributed SPARC Cronus LAN
8. Distributed Multimax Cronus WAN

Figure 5: Analysis Objectives

available more than one version of the code so that we could compare the software overhead of distributed or parallel computing.

Analysis of system-wide performance can be bolstered by analyses of individual performance components. These can be particularly important if one is looking for areas in which to improve the overall system-wide performance.

One important area is the communication vs. computation ratio. We found it particularly useful to analyze our application in terms of a communication/computation profile. For this, we subdivided our application into components which would have different communication/computation ratios based upon the algorithms used or which had different degrees of parallelism . Again the unit of measure in comparing different parts of the profile was elapsed time. We found this profile was useful both in tuning our code for performance and in extrapolating how the performance of the application would scale in terms of size. This analysis supersedes the more traditional measurements of parallel scaling.

Another important area is load balancing. This can be measured as the percent of elapsed time that any node has to wait for other nodes at a synchronization point. Finally, some measurements may need to be taken of the communication traffic on the wide area network. Although interference from other users of the processors or local

area networks can quite likely be controlled since they are locally "owned", interference over a wide area network is quite likely not controllable. Thus, it may be important to establish the performance of the communication under a variety of network loadings. Hardware devices like the HP LANPAs can be used to collect time stamps of the packets.

Experimental Results

As the final phase of our project, the application was run on the distributed-parallel platform established between Rome Laboratory and Syracuse University. The clocks were synchronized between the two Multimaxes, the application code was initiated on the two Multimax nodes by Cronus, and time stamps were recorded by the application and by the two LANPAs.

Although data analysis was not specified as a task of the project, some preliminary analysis was done of system-wide performance based on the elasped times of communication/computation profile. In fact, based on the times of the initial demonstration, some performance tuning was done on the code. We successfully shortened the communication phase by sending groups of tracks together, i.e. we switched to fewer, longer messages. We also adjusted some parts of the program which had minimal amounts of parallelism as to whether they actuallly ran on parallel processors or ran sequentially.

The communication/computation profile that we devised for our application was very simply divided in terms of the main subroutines. Both the master routine and the batch routine have some sequential parts and some minimally parallel parts, by which we mean that the procedures that can be run in parallel are so short as to barely make it worthwhile to incur the overhead of parallel invocation. The balance routine has all the message-passing of the whole program; it's times are overwhelmingly communication. Finally, the "nl_filt" routine has the main computation of the precision filter; it has substantial parallelism in terms of the amount of computation per parallel invocation. All the possible parallel routines have a grain size much larger than the total number of processors of the two machines.

We tested our programs with a number of data sets which varied in the number of missiles to be traced, the timing sequence in which the missiles were launched, etc. However, relative timings did not vary significantly over these data sets, so we chose one data set to report on here. A profile of this data set over 100 iterations of programs is shown in Figure 6. In this data set, the bulk of the new missile tracks occurs between iterations 10 and 25, yielding the peak of the tracks file, which contains all potential tracks. As tracks are filtered, the number of tracks settles down by iteration 65 to the correct number of missiles, around 680. After that time, tracks are dropped by the program as they enter the "post-boot phase", which this program does not process.

The first set of timing tests that we ran was to compare sequential versus parallel. In

fact, we varied the number of processors on one parallel machine from 1 to 16 and measured traditional parallel speedups. An example of these results is shown in Figure 7. In this graph, the running time of all procedures is totalled at each iteration step. The biggest speedup occurs between one processor and two processors, where the total time is almost halved. Good speedups continue as processors are added until five processors and then not more speedups occur. Suprisingly, this application only has useful parallelism for up to five processors.

Since we were surprised at this result, we did extensive timings on the individual procedures of the applications. In particular, we found that this limit on parallelism was also true of the nl_filt procedure, which is the main computational module and should be parallel in the number of tracks. In Figure 8, we show the cumulative running time of this module, that is, at each iteration we add in that iteration's time to a running total. This, again, clearly shows the limitation of five processors, which again points up the overhead limitations imposed by process invocation in this software model.

We proceeded to the timings of the main goal of our project, which was to compare all the processing modes: sequential, distributed, parallel, and distributed-parallel. In Figure 9, these results are shown on a graph of cumulative running times for all procedures up to 100 iterations. We limited the parallel case to six processors and the distributed parallel case (marked DP in the figure) to six processors on each of the two distributed machines, in view of our previous result on the parallel overhead limitation. Interestingly, the distributed-parallel case still shows substantial improvement over the parallel case, although it is far from being twice as fast. We noted that in the distribution process, which is also true in the distributed versus sequential cases, the tracks file is divided in half and many of the task's applications, such as comparing a track to all other local tracks, were also halved in time. Offsetting this gain, of course, was the time spent in dividing the tracks and in communicating them to the other machine.

Finally, we would like to show some results regarding the communication/computation profile. For this, it is instructive to view several different problem sizes. In Figures 10, 11, and 12, we show cumulative elasped times for the different modules in different computing modes for problem sizes of 130, 385, and 680 targets, respectively. The cumulative elasped times are shown on a non-uniform scale so that the communication/computation ratios are shown for each problem size. From theses figures, we see that the percentage of time spent in the communication phase, contained in the balance routines, tend to decrease as the problem size gets larger. Although this is too small a collection to infer continuing decreases, it certainly indicates hope for successful distributed implementations of very large problems of this type.

Conclusions

This project certainly demonstrates the feasibility of distributed-parallel computing for an interesting class of applications. No precise performance judgements can be given

based on such a limited investigation, but much promise is shown for the future. All of the components of this system — the Encore Multimax processors, the T1 line, and the software — are known to be slower than the components that will be available in the near future. This project shows no reason not to believe that the increased performance shown on this platform cannot scale to future faster systems.

References

Huy T. Cao and Clive F. Baillie, Caltech Missile Tracking Program A Benchmark Comparison: Ncube and T800 vs. Sequent Balance and Symmetry, Caltech report, C3P 673, October 1988.

P. Messina, C. Baillie, E. Felten, P. Hipes, and R. Williams, Benchmarking advanced architecture computers, *Concurrency: Practice and Experience*, vol. 2(3), September 1990.

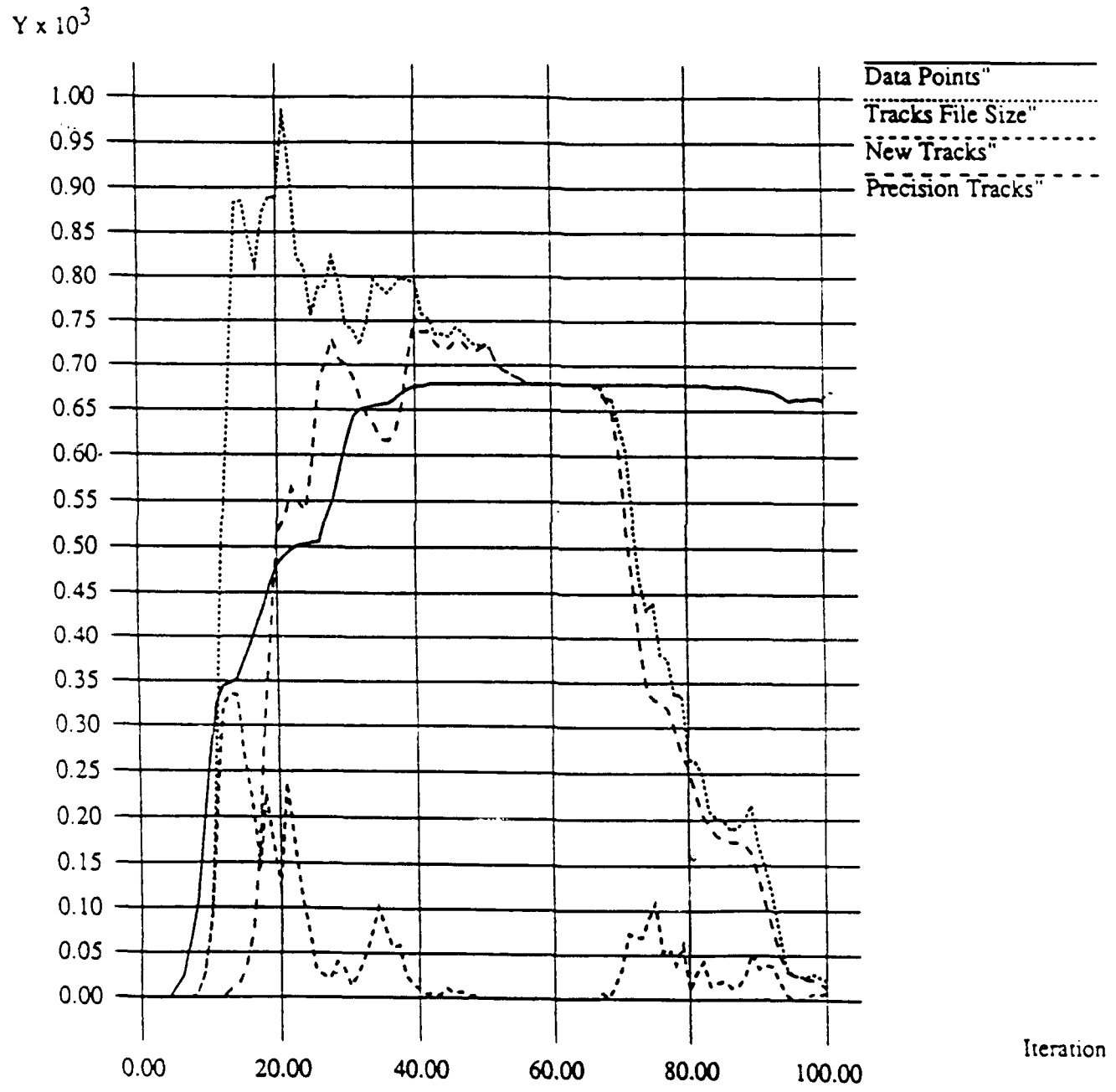


Figure 6. Data Set Profile

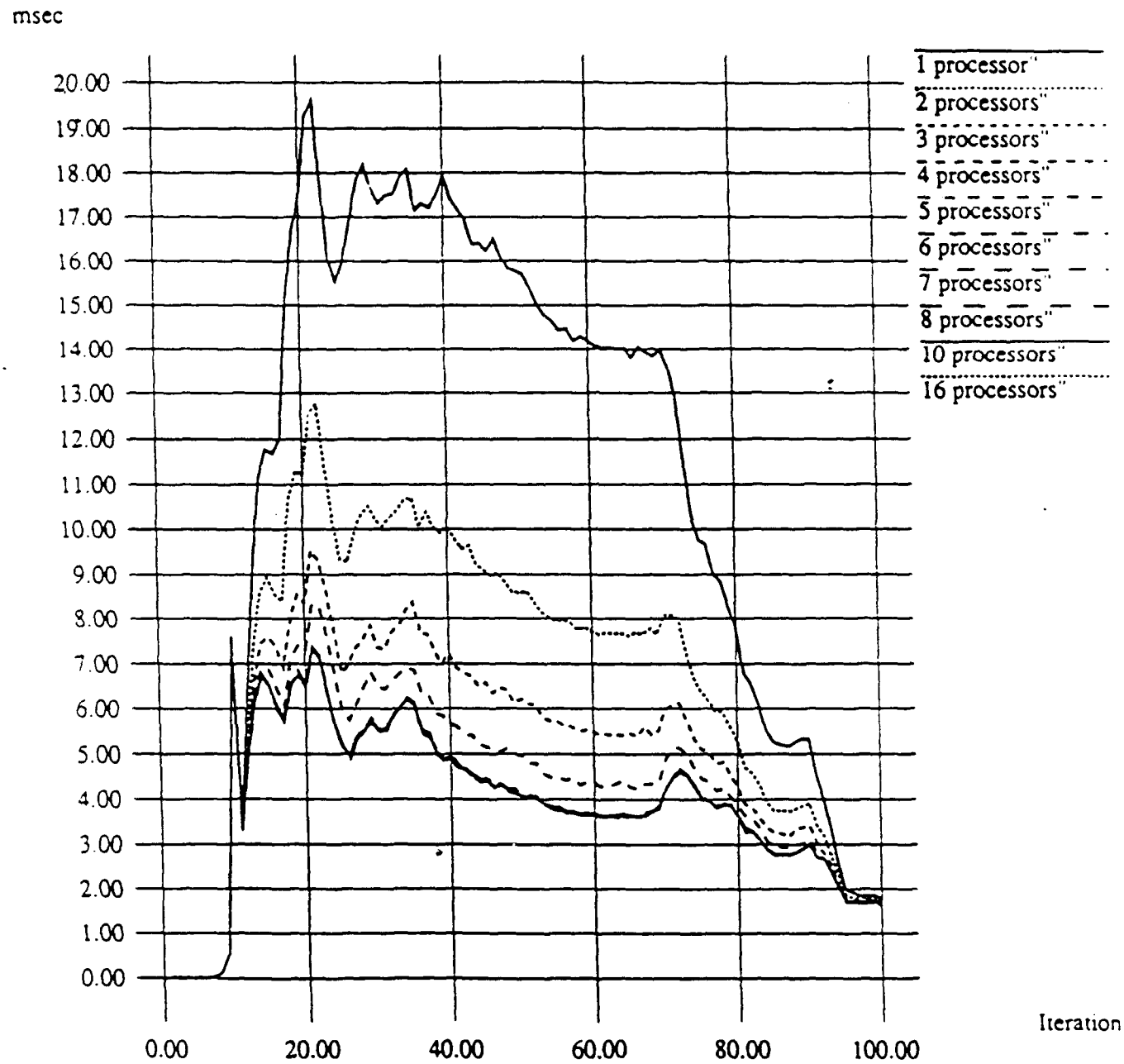


Figure 7:
Parallel Speedup: Total running time of all procedures per iteration

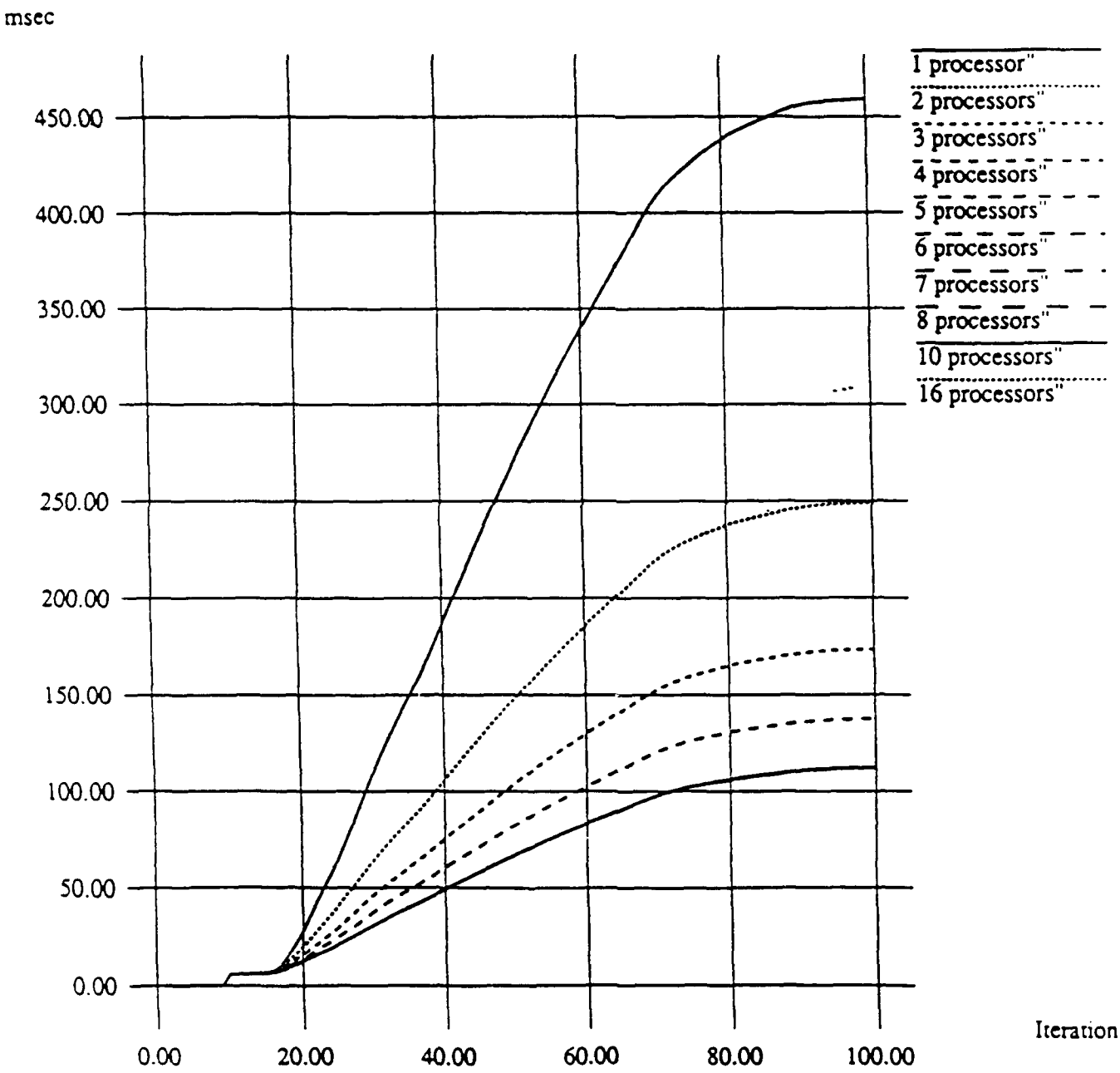


Figure 8:
Cumulative time of the procedure nl_filt up to 100 iterations

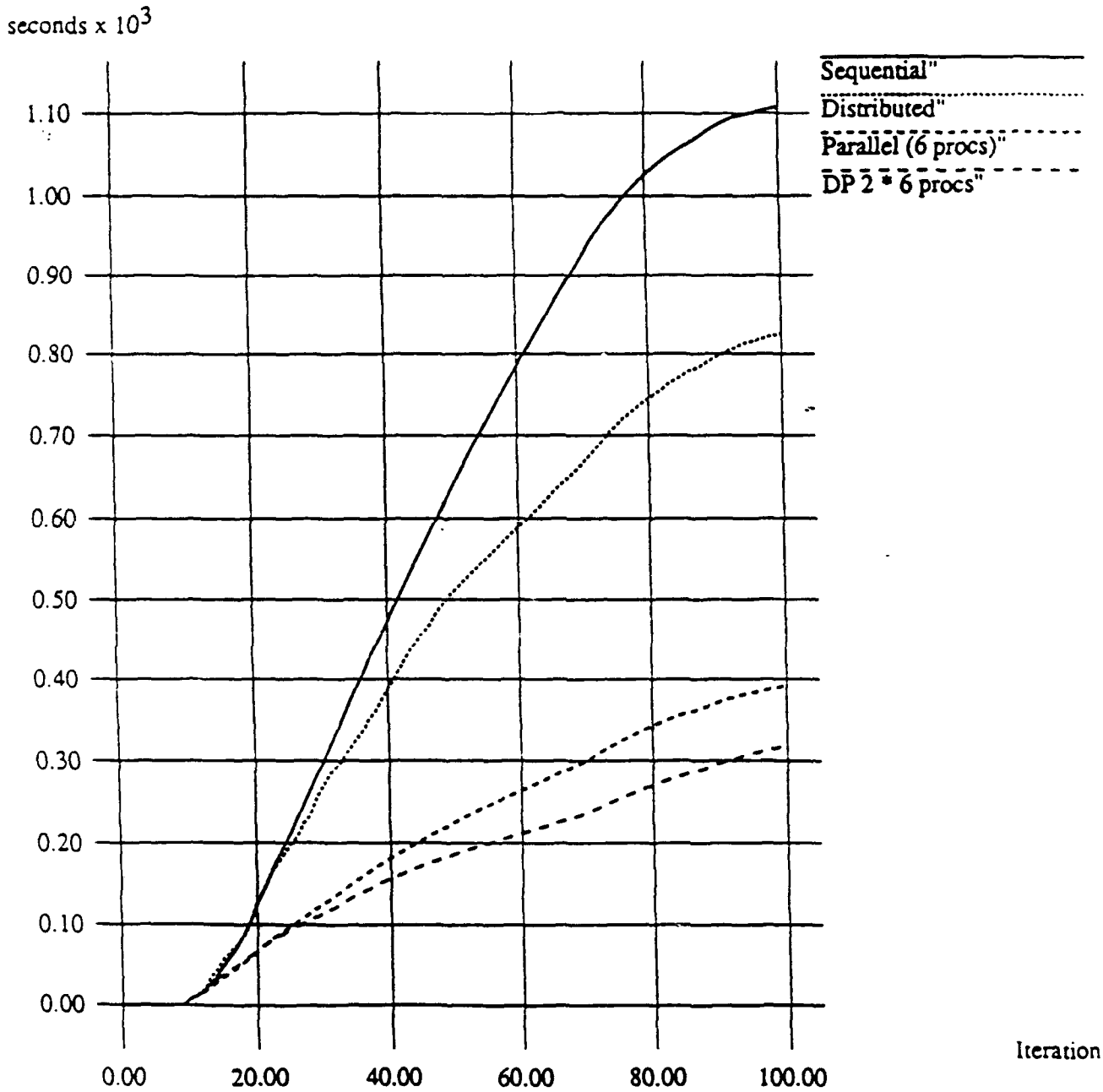


Figure 9:
Mode Profile: Cumulative time for all procedures shown for each processing mode.

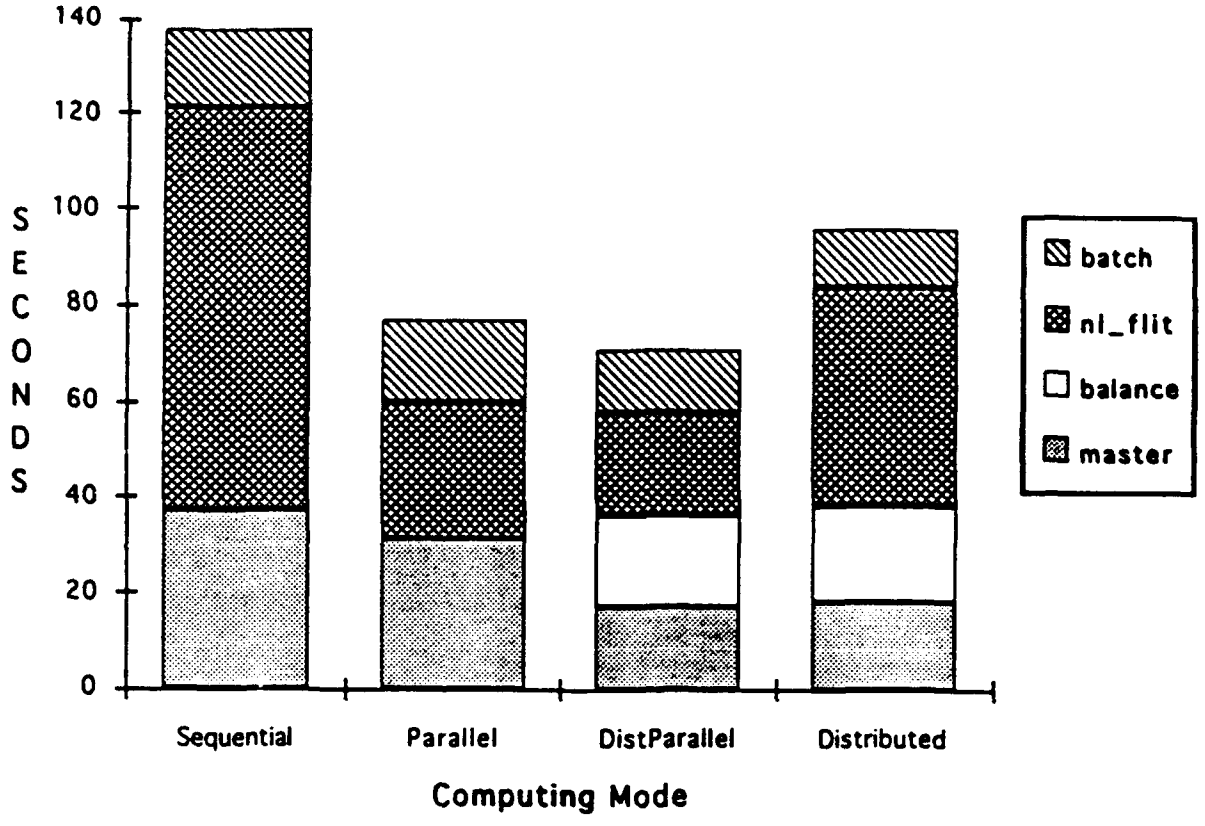


Figure 10:
Elasped time for program modules on a small sized problem

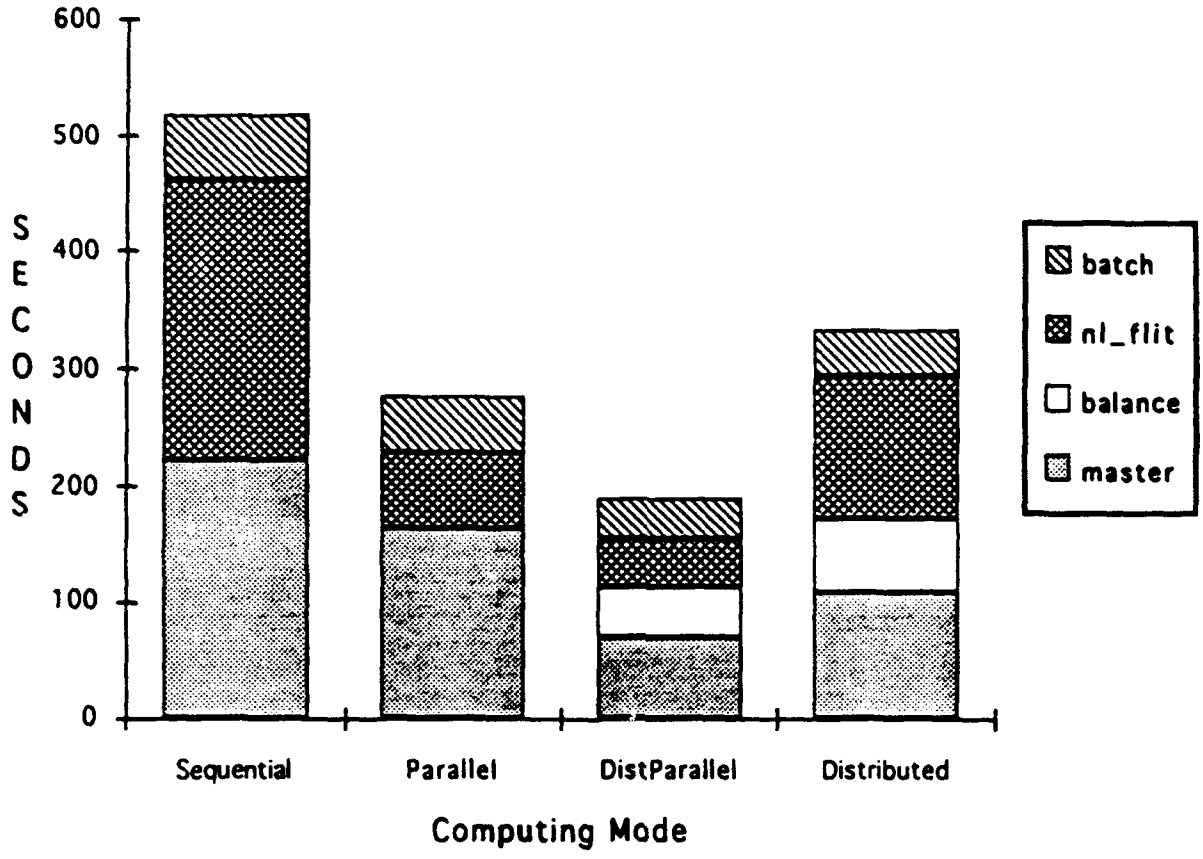


Figure 11:
Elasped time for program modules on a medium sized problem

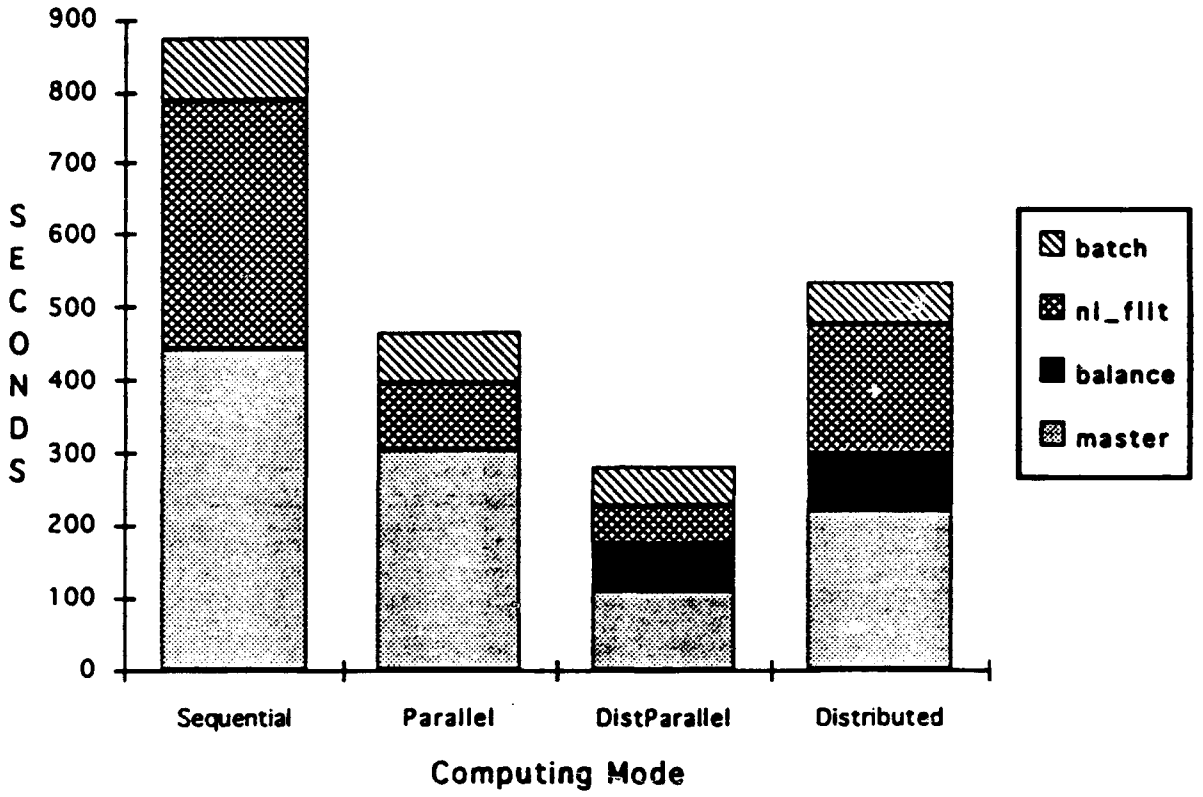


Figure 12:
Elasped time for program modules on a large sized problem

MISSION

OF

ROME LABORATORY

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C3I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESC Program Offices (POs) and other ESC elements to perform effective acquisition of C3I systems. In addition, Rome Laboratory's technology supports other AFMC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.